APPENDIX

ONE-VARIABLE WORD EQUATIONS IN LINEAR TIME

ARTUR JEŻ

Abstract. In this paper we consider word equations with one variable (and arbitrary many appearances of it). A recent technique of recompression, which is applicable to general word equations, is shown to be suitable also in this case. While in general case it is non-deterministic, it determinises in case of one variable and the obtained running time is $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of appearances of the variable in the equation. This matches the previously-best algorithm due to Dąbrowski and Plandowski. Then, using a couple of heuristics as well as more detailed time analysis the running time is lowered to $\mathcal{O}(n)$ in RAM model. Unfortunately no new properties of solutions are shown.

1. Introduction

1.1. **Word equations.** The problem of satisfiability of word equations was considered as one of the most intriguing in computer science and its study was initiated by Markow already in the '50. The first algorithm for it was given by Makanin [13], despite earlier conjectures that the problem is in fact undecidable. The proposed solution was very complicated, both in terms of proof-length, algorithm and computational complexity. It was improved several times, however, no essentially different approach was proposed for over two decades.

An alternative algorithm was proposed by Plandowski and Rytter [19], who presented a very simple algorithm, which in essence guessed a compressed representation of a solution. The running time of this nondeterministic algorithm was polynomial in n and $\log N$, where N is the length of the length-minimal solution. However, at that time the only bound on such length followed from Makanin's work (with further improvements) and it was triply exponential in n.

Soon after Plandowski showed, using novel factorisations, that N is at most doubly exponential [16], showing that satisfiability of word equations is in NEXPTIME. Exploiting the interplay between factorisations and compression he improved the algorithm so that it worked in PSPACE [17].

Producing a description of all solutions of a word equation, even when a procedure for verification of its satisfiability is known, proved to be also of non-trivial task. Still, it is also possible to do this in PSPACE [18], though insight and non-trivial modifications to the earlier procedure are needed.

On the other hand, it is only known that the satisfiability of word equations is NP-hard.

1.1.1. *Two variables.* Since in general the problem is outside P, it was investigated, whether some subclass is feasible, with a restriction on the number of variables being a natural candidate. It was shown by Charatonik and Pacholski [2] that indeed, when only two variables are allowed (though with arbitrarily many appearances), the satisfiability can be verified in deterministic polynomial time. The degree of the polynomial was very hight, though. This was improved over the years and the best known algorithm is by Dąbrowski and Plandowski [3] and it runs in $\mathcal{O}(n^5)$ and returns a description of all solutions.

1.1.2. *One variable.* Clearly, the case of equations with only one variable is in P. Constructing a cubic algorithm is almost trivial, small improvements are needed to guarantee a quadratic running time. First non-trivial bound was given by Obono, Goralcik and Maksimenko, who devised an $\mathcal{O}(n \log n)$ algorithm [15]. This was improved by Dąbrowksi and Plandowski [4] to $\mathcal{O}(n + \#_X \log n)$, where $\#_X$ is the number of appearances of the variable in the equation. Furthermore they showed that there are at most $\mathcal{O}(\log n)$ distinct solutions and at most one infinite family of solutions. Intuitively, the $\mathcal{O}(\#_X \log n)$ summand in the running time comes from the time needed to find and test these $\mathcal{O}(\log n)$ solutions.

1998 *ACM Computing Classification System.* F.2.2 [Analysis of Algorithms and Problem Complexity]: Nonnumerical Algorithms and Problems, F.4.3 [Mathematical Logic and Formal Languages]: Formal Languages.

Key words and phrases. Word equations, string unification, one variable equations.

This work was supported by Alexander von Humboldt Foundation.

This work was not completely model-independent, as it assumed that the alphabet Σ is finite or that it can be identified with numbers. A more general solution was presented by Laine and Plandowski [11], who improved the bound on the number of solutions to $\mathcal{O}(\log \#_X)$ (plus the infinite family) and gave an $\mathcal{O}(n \log \#_X)$ algorithm that runs in a pointer machine model (i.e. letters can be only compared and no arithmetical operations on them are allowed); roughly one candidate for the solution is found and tested in linear time.

1.2. **Recompression.**

Recently, the author proposed a new technique of *recompression* based on previous techniques of Mehlhorn et. al[14] (for dynamic text equality testing), Lohrey and Mathissen [12] (for fully compressed membership problem for NFAs) and Sakamoto [20] (for construction of the smallest grammar for the input text). This method was successfully applied to various problems related to grammar-compressed strings [5, 6, 8]. Unexpectedly, this approach was also applicable to word equations, in which case alternative proofs of many known results were obtained using a unified approach [7].

The technique is based on iterative application of two replacement schemes performed on the text t:

pair compression of ab: For two different letters a, b such that substring ab appears in t replace each of ab in t by a fresh letter c.

a's block compression: For each maximal block a^ℓ, where a is a letter and $\ell > 1$, that appears in t, replace all a^ℓs in t by a fresh letter a_ℓ.

In one phase, pair compression (block compression) is applied to all pairs (blocks, respectively) that appeared at the beginning of this phase. Ideally, each letter is then compressed and so the length of t halves, in a worst-case scenario during one phase t is still shortened by a constant factor.

The surprising property is that such a schema can be efficiently applied even to grammar-compressed data [5, 6] or to text given in an implicit way, i.e. as a solution of a word equation [7]. In order to do so, local changes of the variables (or nonterminals) are needed: X is replaced with $a^\ell X$ (or Xa^ℓ), where a^ℓ is prefix (suffix, respectively) of the substitution for X. In this way the a solution that substitutes $a^\ell w$ for X is implicitly replaced with one that substitutes w.

1.2.1. *Recompression and one-variable equations.*

Clearly, as the recompression approach works for general word equations, it can be applied also to restricted subclasses. However, while in case of word equations it heavily relies on the nondeterminism, when restricted to instances with one variable it can be easily determinised. Furthermore, a fairly natural implementation has $\mathcal{O}(n + k \log n)$ running time, so the same as the Dąbrowski and Plandowski algorithm [4]; this is presented in Section 3. Furthermore adding a few heuristics ,data structures as well as applying a more sophisticated analysis yields a linear running time, this is described in Section 4.

1.3. **Outline of the algorithm.**

In this paper we present an algorithm for one-variable equation based on the recompression. It also provides a compact description of all solutions of such an equation. Intuitively: when pair compression is applied, say ab is replaced by c (assuming it *can* be applied) then there is a one-to-one correspondence of the solutions before and after the compression, this correspondence is simply exchange of all abs by cs and vice-versa. The same applies to the block compression. On the other hand, the modification of X can lead to loss of solutions (note that for technical reasons we do note consider the solution $S(X) = \epsilon$): when X is to be replaced with $a^\ell X$ the new equation has corresponding solutions for S *other than* $S(X) = a^\ell$. So before the replacement, it is tested, whether $S(X) = a^\ell$ is a solution and if so, it is reported. The test itself is simple: both sides of the equation are read and their values under substitution $S(X) = a^\ell$ are created on the fly and compared symbol by symbol, until a mismatch is found or both strings end.

It is easy to implement the recompression so that one phase takes linear time. Then the cost can be distributed to explicit words between the variables, each of them is charged proportionally to its length. Consider such a string w, if it is long enough, its length decreases by a constant factor in one phase, see Lemma 8. Thus, the cost of compressing this fragment and testing a solution can be charged to the lost length. However, this is not true when w is short and the $\#_X \log n$ summand in the running time comes from bounding the running time for such 'short' strings.

In Section 4 it is shown that using a couple of heuristics as well as more involved analysis the running time can be lowered to $\mathcal{O}(n)$. The mentioned heuristics are as follows:

- The problematic 'short' words between the variables need to be substrings of the long words, this allows smaller storage size and consequently faster compression.
- when we compare $Xw_1Xw_2 \ldots w_mX$ from one side of the equation with its copy appearing on the other side, we make such a comparison in $\mathcal{O}(1)$ time (using suffix arrays);
- $(S(X)u)^m$ and $(S(X)u')^{m'}$ (perhaps offsetted) are compared in $\mathcal{O}(|u| + |u'|)$ time instead of naive $\mathcal{O}(m \cdot |u| + m \cdot |u|)$, using simple facts from combinatorics on words.

Furthermore a more insightful analysis shows that problematic 'short' words in the equation invalidate several candidate solutions fast. This allows a tighter estimation of the time spent on testing the solutions.

A note on model. In order to perform the recompression efficiently, some algorithm for grouping pairs is needed. When we can identify the symbols in Σ with consecutive numbers, the grouping can be done using RadixSort in linear time. Thus, all (efficient) applications of recompression technique make such an assumption. On the other hand, the second of the mentioned heuristics craves checking string equality in costant time, to this end a suffix array [9] plus a structure for answering *longest common prefix query* (lcp) is employed [10] on which we use range minimum queries [1]. The last structure needs the flexibility of the RAM model to run in $\mathcal{O}(1)$ time per query.

2. Preliminaries

2.1. **One-variable equations.** Consider a word equation $\mathcal{A} = \mathcal{B}$ over one variable X. Then without loss of generality one of $\mathcal{A}$ and $\mathcal{B}$ begin with a variable and the other with a letter:

- if they both begin with the same symbol (be it letter or nonterminal), we can remove this symbol from them, without affecting the set of solutions,
- if they begin with different letters, this equation clearly has no solution.

The same applies to the last symbols of U and V. Thus, in the following we assume that the equation is of the form

(1) $$A_0XA_1 \ldots A_{n_\mathcal{A}-1}XA_{n_\mathcal{A}} = XB_1 \ldots B_{n_\mathcal{B}-1}XB_{n_\mathcal{B}} \ ,$$

where $A_i, B_i \in \Sigma^*$ and $n_\mathcal{A}$ ($n_\mathcal{B}$) denote the number of X appearances in $\mathcal{A}$ ($\mathcal{B}$, respectively). Note that exactly one of $A_{n_\mathcal{A}}$, $B_{n_\mathcal{B}}$ is empty and A_0 is non-empty. If this condition is violated for any reason, we greedily repair by cutting letters from appropriate strings. We say that A_0 is the *first* word of the equation and the non-empty of $A_{n_\mathcal{A}}$ and $B_{n_\mathcal{B}}$ is the *last word*.

A *substitution* S assigns a string to X, we expand it to $(X \cup \Sigma)^*$ with an obvious meaning. A *solution* is a substitution such that $S(\mathcal{A}) = S(\mathcal{B})$. For a given equation $\mathcal{A} = \mathcal{B}$ we are looking for a description of all its solutions. We treat the empty solution $S(X) = \epsilon$ in a special way and always assume that $S(X) \neq \epsilon$.

Note that if $S(X) \neq \epsilon$, then using (1) we can always determine the first (a) and last (b) letter of $S(X)$ in $\mathcal{O}(1)$ time. In fact, we can determine the length of the a-prefix and b-suffix of $S(X)$.

Lemma 1. *If $A_0 \in a^*$ then $S(X) \in a^*$ for each solution S of $\mathcal{A} = \mathcal{B}$.*

If the first letter of A_0 is a and $A_0 \notin a^*$ then there is at most one solution $S(X) \in a^*$, existence of such a solution can be tested (and its length returned) in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. Furthermore, for $S(X) \notin a^*$ the lengths of the a-prefixes of $S(X)$ and A_0 are the same.

Proof. Consider the case when $A_0 \in a^+$ and suppose that $S(X) \notin a^*$, let $\ell \geq 0$ be the length of the a-prefix of $S(X)$. The length of the a-prefix of $S(\mathcal{A})$ is then $|A_0| + \ell > \ell$, which is the length of the a-prefix of $S(\mathcal{B})$, contradiction. Hence $S(X) \in a^*$.

Consider now the case when A_0 begins with a but $A_0 \notin a^*$, let its a-prefix has length ℓ_A. Consider $S(X) \in a^+$, say $S(X) = a^\ell$. Let the first letter other than a in $\mathcal{B}$ be the $\ell_B + 1$ letter in $\mathcal{B}$ and let it be in block B_i. If there is no such B_i then there is no solution $S(X) \in a^+$, as then $S(\mathcal{B})$ consists only of as, which is not tue for $S(\mathcal{A})$. The length of the a-prefix of $S(\mathcal{A})$ is ℓ_A, while the length of the a-prefix of $S(\mathcal{B})$ is $\ell_B + i \cdot \ell$. Hence $\ell = \frac{\ell_A - \ell_B}{i}$ and this is the only candidate for the solution.

It is easy to verify whether $S(X) = a^\ell$ is a solution for a single ℓ in linear time. It is enough to compare $S(\mathcal{A})$ and $S(\mathcal{B})$ letter by letter, note that thy can be created on the fly while reading $\mathcal{A}$ and $\mathcal{B}$. Each such comparison consumes one symbol from $\mathcal{A}$ and $\mathcal{B}$ (note that if we compare a suffix of $S(X)$, i.e. some $a^{\ell'}$ for $\ell' < \ell$, with $S(X) = a^\ell$ we simply remove $a^{\ell'}$ from both those strings). so the running time is linear.

Lastly, consider $S(X) \notin a^*$. Then the a-prefix of $S(\mathcal{A})$ has length ℓ_A and as $S(X) \notin a^+$, the a-prefix of $S(\mathcal{B})$ is the same as the a-prefix of $S(X)$, which consequently has length ℓ_A. □

Symmetric version of Lemma 1 holds for for the suffix of $S(X)$.

By TestSimpleSolution(a) we denote a procedure, described in Lemma 1, that for $A_0 \notin a^*$ establishes the unique possible solution $S(X) = a^\ell$, tests it and returns ℓ if this indeed is a solution.

2.2. **Representation of solutions.** Consider any solution S of $\mathcal{A} = \mathcal{B}$. We claim that $S(X)$ is uniquely determined by its length and so when describing solution of $\mathcal{A} = \mathcal{B}$ it is enough to give their lengths. If $|S(X)| \leq |A_0|$ then $S(X)$ is a prefix of A_0, so $S(X)$ is uniquely determined by its length. When $|S(X)| > |A_0|$ then $S(\mathcal{A})$ begins with $A_0 S(X)$ while $S(\mathcal{B})$ begins with $S(X)$ and thus $S(X)$ has a period A_0. Consequently, it is of the form $A_0^k A$, where A is a prefix of A_0. So also in this case $S(X)$ is uniquely determined by its length.

Each letter in the current instance of our algorithm represents some string (in a compressed form) of the input equation, we store its *weight* which is the length of such a string. Furthermore, when we replace X with $a^\ell X$ (or $X a^\ell$) we keep track of the weight of a^ℓ. In this way, for each solution of the current equation we know what is the length of the corresponding solution of the original equation.

2.3. **Recompression.** We recall here the technique of recompression [5, 6, 7], restating all important facts about it. Note that in case of one variable many notions simplify.

2.3.1. *Preserving solutions.* All subprocedures of the presented algorithm should preserve solutions, i.e. there should be a one-to-one correspondence between solution before and after the application of the subprocedure. However, as some of the replace X with $a^\ell X$ (or $X b^r$), some solutions may be lost in the process and so they should be reported. We formalise these notions.

We say that a subprocedure *preserves solutions* when given an equation $\mathcal{A} = \mathcal{B}$ it returns $\mathcal{A}' = \mathcal{B}'$ such that for some strings u and v (calculated by the subprocedure)

- some solutions of $\mathcal{A} = \mathcal{B}$ are reported by the subprocedure,
- for each unreported solution S of $\mathcal{A} = \mathcal{B}$ there is a solution S' of $\mathcal{A}' = \mathcal{B}'$, where $S(X) = uS'(X)v$ and $S'(\mathcal{A}') = uS(\mathcal{A})v$
- for each solution S' of $\mathcal{A}' = \mathcal{B}'$ the $S(X) = uS'(X)v$ is an unreported solution of $\mathcal{A} = \mathcal{B}$.

By $\mathrm{PC}_{ab \to c}(w)$ we denote the string obtained from w by replacing each ab by c (we assume that $a \neq b$, so this is well-defined), this corresponds to pair compression. We say that a subprocedure *properly implements pair compression* for ab, if it satisfies the conditions for preserving solutions above, but with $\mathrm{PC}_{ab \to c}(S(X)) = uS'(X)v$ and $\mathrm{PC}_{ab \to c}(S(\mathcal{A})) = uS'(\mathcal{A}')v$ replacing $S(X) = uS'(X)v$ and $\mathcal{A} = u\mathcal{A}v$. Similarly, by $\mathrm{BC}_a(w)$ we denote a string with maximal blocks a^ℓ replaced by a_ℓ and we say that a subprocedure *properly implements blocks compression* for a letter a.

Given an equation $\mathcal{A} = \mathcal{B}$, its solution S and a pair $ab \in \Gamma^2$ appearing $S(U)$ (or $S(V)$) we say that this appearance is *explicit*, if it comes from substring ab of U (or V, respectively); *implicit*, if it comes (wholly) from $S(X)$; *crossing* otherwise. A pair is *crossing* if it has a crossing appearance and *non-crossing* otherwise. Similar notion applies to maximal blocks of as, in which case we say that a *has a crossing block* or it *has no crossing blocks.* Alternatively, a pair ab is crossing if b is the first letter of $S(X)$ and aX appears in the equation or a is the last letter of $S(X)$ and Xb appears in the equation or a is the last and b the first letter of $S(X)$ and XX appears in the equation.

Unless explicitly stated, we consider crossing/non-crossing pairs ab in which $a \neq b$. Note that as the first (last) letter of $S(X)$ is the same for each S, the definition of the crossing pair does not depend on the solution; the same applies to crossing blocks.

When a pair ab is non-crossing, its compression is easy, as it is enough to replace each explicit ab with a fresh letter c

Algorithm 1 PairCompNCr(a, b) Pair compression for a non-crossing pair

```
1: let $c \in \Gamma$ be an unused letter
2: replace each explicit $ab$ in $\mathcal{A}$ and $\mathcal{B}$ by $c$
```

Similarly when none block of a has a crossing appearance, the a's blocks compression consists simply of replacing explicit a blocks.

Algorithm 2 BlockCompNCr(a) Block compression for a letter a with no crossing block

```
1: for each explicit $a$'s $\ell$-block appearing in $U$ or $V$ do
2:     let $a_\ell \in \Gamma$ be an unused letter
3:     replace every explicit $a$'s $\ell$-block appearing in $\mathcal{A}$ or $\mathcal{B}$ by $a_\ell$
```

Lemma 2. *Let ab be a non-crossing pair then* PairCompNCr(a, b) *properly implements the pair compression for ab.*

Let a has no crossing blocks, then BlockCompNCr(a) *properly implements the block compression for a.*

Proof. Consider first the case of PairCompNCr. Suppose that $\mathcal{A} = \mathcal{B}$ has a solution S. Define S': $S'(X)$ is equal to $S(X)$ with each ab replaced with c (where c is a new letter). Consider $S(\mathcal{A})$ and $S'(\mathcal{A}')$. Then $S'(\mathcal{A}')$ is obtained from $S(\mathcal{A})$ by replacing each ab: the explicit appearances of ab are replaced by PairCompNCr(a, b), the implicit ones are replaced by the definition of S' and by the assumption there are no crossing appearances. The same applies to $S(\mathcal{B})$ and $S'(\mathcal{B}')$. Hence $S'(\mathcal{A}') = \mathrm{PC}_{ab\to c}\, S(\mathcal{A}) = \mathrm{PC}_{ab\to c}\, S(\mathcal{B}) = S'(\mathcal{B}')$. The proof in the other direction is the same: for S' we define S such that $S(X)$ is obtained form $S(X)$ by replacing each c by ab. It can be easily shown that $S(\mathcal{A}) = S(\mathcal{B})$ for such an S, furthermore $S'(\mathcal{A}') = \mathrm{PC}_{ab\to c}\, S(\mathcal{A})$ and $S'(\mathcal{B}') = \mathrm{PC}_{ab\to c}\, S(\mathcal{B})$.

The proof for the block compression follows in the same way. ☐

The main idea of the recompression method is the way it deals with the crossing pairs: imagine that ab is a crossing pair, this is because $S(X) = bw$ and aX appears in $\mathcal{A} = \mathcal{B}$ or $S(X) = wa$ and bX appears in it (the remaining case, in which $S(X) = awb$ and XX appears in the equation is treated in the same way). The cases are symmetric, so we deal only with the first one. To 'uncross' ab in this case it is enough to 'left-pop' b from X: replace each X in the equation with bX and implicitly change the solution to $S(X) = w$.

Algorithm 3 Pop(a, b)

```
1: if $b$ is the first letter of $S(X)$ then
2:     if TestSimpleSolution($b$) returns 1 then                ▷ $S(X) = b$ is a solution
3:         report solution $S(X) = b$
4:     replace each $X$ in $\mathcal{A} = \mathcal{B}$ by $bX$       ▷ Implicitly change $S(X) = bw$ to $S(X) = w$
5: if $a$ is the last letter of $S(X)$ then
6:     if TestSimpleSolution($a$) returns 1 then                ▷ $S(X) = a$ is a solution
7:         report solution $S(X) = a$
8:     replace each $X$ in $\mathcal{A} = \mathcal{B}$ by $Xa$       ▷ Implicitly change $S(X) = w'a$ to $S(X) = w'$
```

Lemma 3. Pop(a, b) *preserves solutions and after its application the pair ab is noncrossing.*

Proof. We show that ab in $\mathcal{A} = \mathcal{B}$ is noncrossing. Consider whether X was replaced by bX is line 4. If not, then the first letter of $S(X)$ and $S'(X)$ is not b, so ab cannot be crossing because of X prefix. Suppose that X was replaced with bX. Then the letter to the left of X is not a, and so ab cannot be crossing because of X prefix.

A similar analysis is applied to the right-end of X, which yields that ab cannot be a crossing pair.

Observe that each solution reported by Pop is verified, so it is indeed a solution. Furthermore, as only one-letter solutions are returned, they correspond to ϵ solutions of $\mathcal{A}' = \mathcal{B}'$, which are not considered.

Consider first the modification performed by in the if-statement from line 1. If $S(X)$ does not begin with b (recall that all solutions have the same first letter) then nothing changes and the set of solutions of preserved. If $S(X) = bw$ and additionally

$w = \epsilon$: then it is reported in line 3;

$w \neq \epsilon$: then $S'(X) = w$ is a solution of the obtained equation.

The same analysis for the if-statement from line 5 yields the claim of the lemma. ☐

Now the presented procedures can be merged into one procedure that turns crossing pairs into noncrossing ones and then compresses them, effectively compressing crossing pairs.

Algorithm 4 PairComp(a, b) Turning crossing pair ab into non-crossing ones and compressing it

```
1: run Pop$(a, b)$
2: run PairCompNCr$(a, b)$
```

Lemma 4. PairComp(a, b) *properly implements the pair compression of the pair ab.*

The proof follows by combining Lemma 2 and 3.

There is one issue: the number of non-crossing pairs can be large, however, a simple preprocessing, which basically applies Pop, is enough to reduce the number of crossing pairs to 2.

Algorithm 5 PreProc Ensures that there are at most 2 crossing pairs

```
1: let $a$, $b$ be the first and last letter of $S(X)$
2: run Pop$(a, b)$
```

Lemma 5. PreProc *preserves solution and after its application there are at most two crossing pairs.*

Proof. It is enough to show that there are at most 2 crossing pairs, as the rest follows form Lemma 3. Let a and b be the first and last letters of $S(X)$, and a', b' such letters after the application of PreProc. Then each X is proceeded with a and succeeded with b in $\mathcal{A}' = \mathcal{B}'$. So the only crossing pairs are aa' and $b'b$ (note that this might be the same pair). ☐

The problems with crossing blocks can be solved in a similar fashion: a has a crossing block, if and only if aa is a crossing pair. So we 'left-pop' a from X until the first letter of $S(X)$ is different than a, we do the same with the ending letter b. This can be alternatively seen as removing the whole a-prefix (b-suffix, respectively) from X: suppose that $S(X) = a^\ell w b^r$, where w does not start with a nor end with b. Then we replace each X by $a^\ell X b^r$ implicitly changing the solution to $S(X) = w$, see Algorithm 6.

Algorithm 6 CutPrefSuff Cutting prefixes and suffixes

```
1: let $a$ be the first letter of $S(X)$
2: report solution found by TestSimpleSolution$(a)$
3: let $\ell > 0$ be the length of the $a$-prefix of $S(X)$
4: replace each $X$ in $\mathcal{A} = \mathcal{B}$ by $a^\ell X$            ▷ $a^\ell$ is stored in a compressed form,
5:                                                      ▷ implicitly change $S(X) = a^\ell w$ to $S(X) = w$
6: let $b$ be the last letter of $S(X)$
7: report solution found by TestSimpleSolution$(b)$
8: let $r > 0$ be the length of the $b$-suffix of $S(X)$
9: replace each $X$ in $\mathcal{A} = \mathcal{B}$ by $Xb^r$             ▷ $b^r$ is stored in a compressed form,
10:                                                     ▷ implicitly change $S(X) = wb^r$ to $S(X) = w$
```

Lemma 6. CutPrefSuff *preserves solutions and after its application there are no crossing blocks of letters.*

Proof. Consider first only the changes done by the modification of the prefix. Suppose that $S(X) = a^\ell w$, where w does not begin with a. If $w = \epsilon$ then this solution is reported in line 2. Otherwise, the $S'(X) = w$ is the solution of the new equation. Similarly, for any solution $S'(X) = w$ the $S(X) = a^\ell w$ is the solution of the original equation.

The same analysis can be applied to the modifications of the suffix.

Lastly, suppose that some letter c has a crossing block, without loss of generality assume that c is the first letter of $S(X)$ and cX appears in the equation. But this is not possible: X was replaced by $a^\ell X$ and so the only letter to the left of X is a and $S(X)$ does not start with a, contradiction. ☐

The CutPrefSuff allows defining a procedure BlockComp that compresses maximal blocks of all letters, regardless of whether they have crossing blocks or not.

Algorithm 7 BlockComp Compressing blocks of a

```
1: Letters ← letters appearing in the equation
2: run CutPrefSuff                              ▷ Removes crossing blocks of $a$
3: for each letter $a \in$ Letters do
4:     BlockCompNCr($a$)
```

Lemma 7. BlockComp *properly implements the block compression for letters present in $\mathcal{A} = \mathcal{B}$ before its application.*

The proof follows by combining Lemma 2 and 6.

3. Main algorithm

The following algorithm OneVarWordEq is basically a simplification of the general algorithm for testing the satisfiability of word equations [7].

Algorithm 8 OneVarWordEq Reports solutions of a given one-variable word equation

```
1: while $|A_0| > 1$ do
2:     BlockComp                          ▷ Compress blocks, in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time.
3:     PreProc                            ▷ There are only two crossing pairs, see Lemma 5
4:     Crossing ← list of crossing pairs                   ▷ There are two such pairs
5:     Non-Crossing ← list of non-crossing pairs
6:     for each $ab \in$ Non-Crossing do  ▷ Compress non-crossing pairs, in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)|$
7:         PairCompNCr($a, b$)
8:     for $ab \in$ Crossing do           ▷ Compress the 2 crossing pairs, in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)|$
9:         PairComp($a, b$)
10: TestSolution                          ▷ Test solutions from $a^*$, see Lemma 9
```

We call one iteration of the main loop a *phase.*

Theorem 1. OneVarWordEq *runs in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}| + (n_{\mathcal{A}} + n_{\mathcal{B}}) \log(|\mathcal{A}| + |\mathcal{B}|))$ and correctly reports all solution of a word equation $\mathcal{A} = \mathcal{B}$.*

Before showing the running time, let us first comment on how the equation is stored. Each of sides ($\mathcal{A}$ and $\mathcal{B}$) is represented as a table of pointers to strings, i.e. to A_0, A_1, ..., $A_{n_{\mathcal{A}}}$ and B_0, B_1, ..., $B_{n_{\mathcal{B}}}$. Each of those words is stored as a doubly-linked list. When we want to refer to a concrete word in a phase, we use names A_i and B_j, when we want to stress its evolution in phases, we use names $\mathcal{A}$ i-word and $\mathcal{B}$ j-word.

The most important property of OneVarWordEq is that the explicit strings between the variables shorten (assuming that they have a large enough length).

We say that a word A_i (B_i) is *short* if it consists of at most 100 letters and *long* otherwise. To avoid usage of strange constants and its multiplicities, we shall use $N = 100$ to denote this value.

Lemma 8. *Consider the length of the $\mathcal{A}$ i-word (or $\mathcal{B}$ j-word). If it is long then its length is reduced by $1/4$ in this phase. If it is short then after the phase it still is. The length of each unreported solution is reduced by at least $1/4$ in a phase.*

Additionally, if the first (last) word is short then its length is shortened by at least 1 in a phase.

Proof. We shall first deal with the words and then comment how this argument extends to the solutions. Consider two consecutive letters a, b in any word at the beginning of a phase. We show that at least one of those letters is compressed in this phase:

$a = b$**:** Then they are compressed using BlockComp.

$a \neq b$**:** If one of them is compressed by BlockComp then we are done, so suppose not. Then ab is a pair appearing in the equation and as such we try to compress it, either in line 7 or in line 9. This appearance cannot be compressed only when one of the letters a, b was already compressed, in some other pair.

This means that in a word of length k during the phase at least $\frac{2(k-1)}{3}$ letters are compressed (since we can associate with each uncompressed letter the two neighbouring compressed letters, moreover, one compressed letter is associated with only one uncompressed letter) i.e. its length is reduced by at least $\frac{k-1}{3}$ letters.

On the other hand, letters are introduced into words by popping them from variables. Let *symbol* denote a single letter or block a^ℓ that is popped into a word, we investigate, how many symbols are introduced in this way in one phase. At most one symbol is popped to the left and one to the right by BlockComp in line 1, the same holds for PreProc in line 3 Moreover, one symbol is popped to the left and one to the right in line 7; since this line is executed twice, this yields 8 symbols in total. Note that the symbols popped by BlockComp are replaced by a single letter, so the claim in fact holds for letters and not symbols.

So, consider any word $A_i \in \Gamma^*$ (the proof for B_j is the same), at the beginning of the phase and let A'_i be the corresponding word at the end of the phase. There were at most 8 symbols introduced into A'_i (some of them might be compressed later). On the other hand, we already established that at least $\frac{|A_i|-1}{3}$ letters were removed due to compression. Hence

$$|A'_i| \leq |A_i| - \frac{|A_i|-1}{3} + 8 \leq \frac{2|A_i|}{3} + 8\frac{1}{3} \ .$$

It is easy to check that when A_i is short, i.e. $|A_i| \leq N = 100$, then A'_i is short as well and when A_i is long, i.e. $|A_i| > N$ then $|A'_i| \leq \frac{3}{4}|A_i|$.

It is left to show that the first word shortens by at least one letter in each phase. Consider that if a letter a is left-popped from X then the same letter is removed from the front of the first word in order to preserve (1). Furthermore the right-popping does not affect the first word at all (as X is not to its right); the same applies to cutting the prefixes and suffixes. Hence the length of the first word is never increased by popping letters. Moreover, if at least one compression (be it block compression or pair compression) is performed inside the first word, its length drops. So consider the first word at the end of the phase let it be A_0. Note that there is no letter representing a compressed pair or block in A_0: consider for the sake of contradiction the first such letter that appeared in the first word. It could not appear through a compression inside the first word (as we assumed that it did not happen), cutting prefixes does not introduce compressed letters, so it could only come through a left-pop from X. But then the first letter of A_0 is the same, contradiction, as this was not the first compressed letter in A_0.

So in A_0 there are no compressed letters. Let ab be the first two letters in A_0. If $a = b$ then they should have been compressed by blocks compression, contradiction. If $a \neq b$ then they were listed either in *Non-Crossing* or *Crossing* and so should have been compressed, contradiction.

Now, consider a solution $S(X)$. We know that $S(X)$ is either a prefix of A_0 or of the form $A_0^\ell A$, where A is a prefix of A_0. In the first case, $S(X)$ is compressed as a substring of a word. In the second observe that argument follows as long as we try to compress the pair between A_0 and A_0. But since the first letter of $S(X)$ and A_0 is the same, this pair is one of the crossing pairs. So, the claim of the lemma holds for $S(X)$ as well. ☐

The correctness of the algorithm follows from Lemmata 7 (for BlockComp), Lemma 5 (for PreProc), Lemma 2 (for PairCompNCr), Lemma 4 (for PairComp) and from the lemma below, which deals with TestSolution.

Lemma 9. *For $a \in \Gamma$ we can report all solutions in which $S(X) \in a^k$ for some natural ℓ in $\mathcal{O}(|\mathcal{A}|+|\mathcal{B}|)$ time. There is either exactly one ℓ for which $S(X) = a^\ell$ is a solution, $S(X) = a^\ell$ is a solution for each ℓ or there is no solution of this form.*

Proof. The construction and proof is similar as in Lemma 1. Suppose that $S(X) = a^\ell$ is a solution of $\mathcal{A} = \mathcal{B}$. We calculate the length of the a-prefix of $S(\mathcal{A})$ and $S(\mathcal{B})$. Consider first letter other than a in $\mathcal{A}$, let it be in the A_{k_A} and suppose that there were ℓ_A letters a before it (if there is non such letter, imagine we attach an 'ending marker' to both $\mathcal{A}$ and $\mathcal{B}$, which then becomes such letter). The clearly the length of the a-prefix of $S(\mathcal{A})$ is $k_A \cdot \ell + \ell_A$. Let additionally $\mathcal{A}'$ be obtained from $\mathcal{A}$ by removing those letters a and variables in between them. Similarly, define k_B, ℓ_B and $\mathcal{B}'$. Then the length of the a-prefix of $S(\mathcal{B})$ is $k_B \cdot \ell + \ell_B$.

The substitution $S(X) = a^\ell$ clearly is a solution if and only if $k_A \cdot \ell + \ell_A = k_B \cdot \ell + \ell_B$ and $S(\mathcal{A}') = S(\mathcal{B}')$. Consider the number of natural solutions of the equation

$$k_A \cdot x + \ell_A = k_B \cdot x + \ell_B :$$

no natural solution: clearly there is no solution of the word equation $\mathcal{A} = \mathcal{B}$;

one solution $x = \ell$: then $S(X) = a^\ell$ is the only possible solution among a^* of $\mathcal{A} = \mathcal{B}$. To verify whether S satisfies $\mathcal{A}' = \mathcal{B}'$ it is is enough to run $\mathsf{TestSimpleSolution}(a)$ on it and see whether it returns ℓ.

satisfied by all natural numbers: then the a-prefixes of $\mathcal{A}$ and $\mathcal{B}$ are of the same length for each $S(X) \in a^*$. We thus repeat the procedure for $\mathcal{A}' = \mathcal{B}'$, shortening them so that they obey the form (1), if needed. Clearly, solutions in a^* of $\mathcal{A}' = \mathcal{B}'$ are exactly the solutions of $\mathcal{A} = \mathcal{B}$ in a^*.

The stopping condition for the recurrence above are obvious: if $\mathcal{A}'$ and $\mathcal{B}'$ are both empty then we are done (each $S(X) = a^\ell$ is a solution of this), if exactly one of them is empty and the other is not, there is no solution at all.

Lastly, observe that the cost of the subprocedure above is proportional to the amount of read letters, which are not then read again, so the running time is $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ □

Lemma 10. *One phase of* OneVarWordEq *can be performed in* $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ *time.*

Proof. For grouping of pairs and blocks we use RadixSort, to this end it is needed that the alphabet of (used) letters can be identified with an interval of numbers, of size $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. In the first phase of OneVarWordEq this follows from the assumption on the input. In fact, this assumption can be weakened a little: it is enough to assume that at the if $\Gamma \subseteq \{1, 2, \ldots, \mathsf{poly}(|\mathcal{A}| + |\mathcal{B}|)\}$: in such case we can use RadixSort to sort Γ in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ and then replace Γ with set of consecutive natural numbers. At the end of this proof we describe how to bring back this property at the end of the phase.

To perform BlockComp we want for each letter a appearing in the equation to have lists of all maximal a-blocks appearing in $\mathcal{A} = \mathcal{B}$ (note that after CutPrefSuff there are no crossing blocks, see Lemma 6). This is done by reading $\mathcal{A} = \mathcal{B}$ and listing triples (a, k, p), where k is the length of a maximal block of as of length k and p is a pointer to the beginning of this appearance. Notice, that the maximal block of a's may consist also of prefixes/suffixes that were cut from X by CutPrefSuff. However, by Lemma 1 such a prefix is of length at most $|A_0| \leq |\mathcal{A}| + |\mathcal{B}|$ (and similar analysis applies for the a suffix). Then each maximal block includes at most one such prefix and one such suffix thus the total length of the a maximal block is $3(|\mathcal{A}|+|\mathcal{B}|)$. Hence, the triples (a, k, p) can be sorted by their first two coordinates using RadixSort in total time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. We go through the list of maximal blocks. For a fixed letter a, we use the pointers to localise a's blocks in the rules and we replace each of its maximal block of length $\ell > 1$ by a fresh letter. Since the blocks of a are sorted, all blocks of the same length are consecutive on the list, and replacing them by the same letter is easily done.

To compress all non-crossing pairs, i.e. to perform the loop in line line 7 we do a similar thing as for blocks: we read both $\mathcal{A}$ and $\mathcal{B}$, whenever we read a pair ab for $a \neq b$, we add a triple (a, b, p) to the temporary list, where p is a pointer to this position. Then we sort all these pairs according to lexicographic order on first two coordinates, we use RadixSort for that. Since in each phase we number the letters appearing in $\mathcal{A} = \mathcal{B}$ using consecutive numbers, this can be done in time $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. The appearances of the crossing pairs can be removed from the list: by Lemma 5 there are at most two crossing pairs and they can be easily established (by looking at $A_0 X A_1$). So we read the sorted list of pairs appearances and we remove from it the ones that correspond to a crossing pair. Lastly, we go through this list and replaces pairs, as in the case of blocks. Note that when we try to replace ab it might be that this pair is no longer there as one of its letters was already replaced, in such a case we do nothing.

We can compress each of the crossing pairs naively in $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time by simply first applying the popping and then reading the equation form the left to the right.

It is left to describe, how to enumerate (with consecutive numbers) letters in Γ at the end of each phase. Firstly notice that we call easily enumerate all letters introduced in this phase and identify them (at the end of this phase) with $\{1, \ldots, m\}$, where m is the number of introduced letters. Next by the assumption the letters in Γ (from the beginning of this phase) are already identified with a subset of $\{1, \ldots, |\mathcal{A}| + |\mathcal{B}|\}$, we want to renumber them, so that the subset of letters from Γ that are present at the end of the phase is identified with $\{m + 1, \ldots, m + m'\}$ for an appropriate m'. Note that each block of a^k for $k > 2$ was replaced, so only letters with maximal block of length 1 may have remained. To identify them we can simply extend the algorithm for block compression: it should also make a separate list in which maximal blocks of length 1 are listed (with pointers to appearances), this list is also sorted according to the letter value. We then go through the list of those letters and when dealing with a we use the pointers to the appearances of a to establish, whether a is still in the current equation $\mathcal{A} = \mathcal{B}$. If so, we assign a the next free number. Clearly the whole process takes at most $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time. ☐

Lemma 11. *As soon as first or last word becomes short, the rest of the running time of* OneVarWordEq *is $\mathcal{O}(n)$.*

Proof. This is fairly simple: one phase takes $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$ time by Lemma 10 (this is at most $\mathcal{O}(n)$ by Lemma 8) and as Lemma 8 guarantees both first word and last word are shortened by at least one letter, there will be at most N many phases. Lastly, Lemma 9 shows that TestSolution also runs in $\mathcal{O}(n)$. ☐

Lemma 12. *The running time of* OneVarWordEq *till one of first or last word becomes short is $\mathcal{O}(n + (n_{\mathcal{A}} + n_{\mathcal{B}}) \log n)$.*

Proof. By Lemma 10 the time of one iteration of OneVarWordEq is $\mathcal{O}(|\mathcal{A}| + |\mathcal{B}|)$. We distribute the cost among the $\mathcal{A}$ words and $\mathcal{B}$ words: we charge $\beta|A_i|$ to $\mathcal{A}$ i-word (similarly to for $\mathcal{B}$ words), for some positive β. Fix $\mathcal{A}$ i-word, we separately estimate how much was charged to it when it was a long and short word.

long: Let n_i be the initial length of this word. Then these costs are at most $\beta n_i + \frac{3}{4}\beta n_i + \frac{3}{4}^2 \beta n_i + \ldots \leq 4\beta n_i$.

short: Since $\mathcal{A}$ i-word is short, its length is at most N, so we charge at most $N\beta$ to it. Notice, that there are $\mathcal{O}(\log n)$ iterations of the loop in total, as first word is of length at most n and it shortens by $\frac{3}{4}$ in each iteration when it is long and we calculate only the cost when it is long. Hence the cost charged in this way is at most $\mathcal{O}(\log n)$, summing over all words yields $\mathcal{O}((n_{\mathcal{A}} + n_{\mathcal{B}}) \log n)$. ☐

4. Heuristics and Better Analysis

The intuition gained from the analysis in the previous section is that the main obstacle in the linear running time is the necessity of dealing with short words, as the time spend on processing them is difficult to charge. This applies to both the compression performed within the short words, which does not guarantee any reduction in length, and to testing of the candidate solutions, which cannot be charged to the length decrease of the whole equation.

The improvement to linear running time is done by four major modifications, which are described in details in the following subsections:

several equations: Instead of a single equation, we store a system of several equations and look for a solution of such a system. This allows removal of some words from the equations and thus decreases the overall storing space and testing time.

small solutions: We identify a class of particularly simple solutions, called *small*, and show that a solution is reported within $\mathcal{O}(1)$ phases from the moment when it became small. In several problematic cases of the analysis we are able to show that the solutions involved are small and so it is easier to charge the time spent on testing them.

storage: The storage is changed so that all words are represented by a structure of size proportional to the size of the long words. In this way the storage space decreases by a constant factor in each phase and so the running time (except for testing) is linear.

testing: The testing procedure is modified, so that the time it spends on the short words is reduced. In particular, we improve the rough estimate that one TestSimpleSolution takes time proportional to the equation to an estimation that actually counts for each word whether it was included in the test or not.

4.1. Suffix arrays and lcp arrays.

A suffix array $SA[1\,..\,m]$ for a string $w[1\,..\,m]$ stores the m non-trivial suffixes of w, that is $w[m], w[m-1\,..\,m], \ldots, w[1\,..\,m]$ in (increasing) lexicographical order. In other words, $SA[k] = p$ if and only if $w[p \ldots m]$ is the k-th suffix according to the lexicographical order. It is known that such an array can be constructed in $\mathcal{O}(m)$ time [9] assuming that RadixSort is applicable to letters, i.e. that they are integers from $\{1, 2, \ldots, m^c\}$ for some constant c.

Using a suffix array the equality testing for substrings of w reduces to the *longest common prefix* (lcp) query: observe that $w[i\,..\,i+k] = w[j\,..\,j+k]$ if and only if the common prefix of $w[i\,..\,m]$ and $w[j\,..\,m]$ is at least k. The first step in constructing a data structure for answering such queries is the LCP array: for each $i = 1, \ldots, m-1$ the $LCP[i]$ stores the length of the longest common prefix of $SA[i]$ and $SA[i+1]$. Given a suffix array, the LCP array can be constructed in linear time [10], however, the linear-time construction of suffix arrays can be in fact extended to return also the LCP array [9]

When the LCP array is supplied, the general longest prefix queries reduce to the range minimum queries: the longest common prexif of $SA[i]$ and $SA[j]$ (for $i < j$) is the minimum among $LCP[i], \ldots, LCP[j-1]$, and so it is enough to have a data structure that answers the queries about the minimum in the range in constant time. Such data structures in general case are known and in case of LCP arrays even simpler construction were given [1].

4.2. **Several equations.** The improved analysis assumes that we do not store a single equation, instead, we store several equations and look for substitutions that simultaneosly satisfy all of them. Hence we have a collection $\mathcal{A}_i = \mathcal{B}_i$ of equations, for $i = 1, \ldots, m$, each of them is of the form described by (1); by $\mathcal{A} = \mathcal{B}$ we denote the whole system of those equations. In particular, each of those equations specifies the first and last letter of the solution, length of the a-prefix and suffix etc., exactly in the same way as it does for a single equation. However, it is enough to use only one of them, say $\mathcal{A}_1 = \mathcal{B}_1$, as if there is any conflict then there is no solution at all. The consistency is not checked, simply when we find out about inconsistency, we terminate immediately.

The system of equations stored by OneVarWordEq is obtained by replacing one equation $\mathcal{A}'_i\mathcal{A}''_i = \mathcal{B}'_i\mathcal{B}''_i$ with equivalent two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ (note that in general the latter two equation are not equivalent to the former one, however, we perform the replacement only when they are). Hence we store words using the same tables as before, simply for each word we additionally keep information whether it is a first/last word in some equation. The first (last) word of $\mathcal{A}_i$ has a link to the last word of $\mathcal{A}_{i-1}$ and $\mathcal{A}_i$ (the first word of $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$, respectively). We also say that A_i (B_j) is first or last if it is in any of the stored equations. The original equation implies a natural order on the equations in the system: among the two equations $\mathcal{A}_i = \mathcal{B}_i$ and $\mathcal{A}_j = \mathcal{B}_j$ the one whose words correspond to earlier words from the input equation is earlier. This order is followed whenever we perform any operations on all words of the equations.

All operations on a single equation introduced in the previous sections (popping letters, cutting prefixes and suffixes, pair compression, blocks compression) generalises easily to a system of equations and they preserve their properties and running times, with the length of a single equation replaced by a sum of lengths of all equations. However, some lemmata require a little care: firstly, PreProc should ensure that there are only two crossing pairs. This is the case, as each X in every equation is replaced by the same aXb and $S(X)$ is the same for all equations, which is the main fact used in the proof of Lemma 5.

Secondly, Lemma 8 ensured that in each phase the length of the first and last word is decreased. Currently the first words in each equation may be different, however, the analysis in Lemma 8 applies to each of them.

4.3. **Small solutions.** We say that a word w represented as $w = w_1 w_2^\ell w_3$ (where ℓ is arbitrary) is *almost periodic*, with *period size* $|w_2|$ and *side size* $|w_1 w_3|$ (note that several such representation may exist, we use this notion for a particular representation that is clear from the context). A substitution S is *small*, if $S(X) = (w)^k v$, where w, v are almost periodic, with period size at most N and side size at most $6N$.

Lemma 13. *Suppose that $S(X)$ is a small solution. There is a constant c such that within c phases the corresponding solution is reported by* OneVarWordEq.

Proof. Note that for sure a substitution is tested when it is reduced to one letter. So it is enough to show that a small solution is reduced to one letter within $\mathcal{O}(1)$ phases.

Intuition is as follows: consider an almost periodic word, represented as $w_1 w_2^\ell w_3$. Ideally, all compressions are done separately on w_1, w_3 and each w_2. In this way we obtain a string $w'_1 w'^\ell_2 w'_3$ and from Lemma 8 it follows that $|w'_i| \leq \frac{3}{4}|w_i|$. After $\mathcal{O}(\log|w_2|)$ steps we obtain a word $w_1 w_2^\ell w_3$ in which w_2 is a single letter, and so in this phase w_i^ℓ is replaced with a single letter. Then, since the length of w_1 and w_3 is at most $6N$, after $\mathcal{O}(1)$ phases this is reduced to a single letter. Concerning the small solution, $uw^k v$ we first make such an analysis for w, when it is reduced to a single letter (after $\mathcal{O}(1)$ phases) after one additional phase $w^k = a^k$ is also reduced to one letter (by BlockComp) and so the

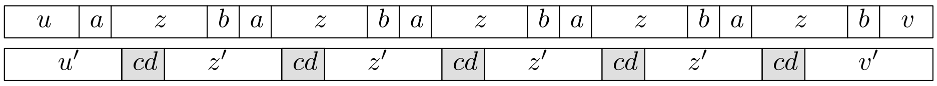

FIGURE 1. The alternative factorisation. The first compressed letters are in grey. For simplicity $\ell = r = 1$. Each z' between the cds is compressed independently.

obtained string $u'a_kv'$ is a concatenation of three almost periodic strings. Using the same analysis as above or each of them we obtain that it takes $\mathcal{O}(1)$ time to reduce them to a single letter.

In reality we need to take into the account that some compression are made on the crossings of the considered strings, however, we can partition the strings so that the result is almost as in the idealised case. We begin with showing an appropriate claim in the borderline cases.

Claim 1. Consider almost periodic substring uw^kv of a solution $S(X)$ with period size at most N and size side at most $6N$. If $k = 1$ or w is a block of letters, this is replaced with one symbol in $\mathcal{O}(1)$ phases.

Proof. If $k = 1$ then $S(X) = uwv$, where $|uvw| \leq 7 \cdot N$ and by Lemma 8 in $\mathcal{O}(1)$ this is reduced to a single-letter solution which is then reported. If w is a block of the same letter then it is replaced with a single symbol in one phase, reducing to the previous case. ☐

We say that for a string uwv during one phase of OneVarWordEq the letters in w are *compressed independently*, if every compressed pair or block were either wholly within this w or wholly outside this w (in some sense this corresponds to the non-crossing compression).

Claim 2. Consider almost periodic substring uw^kv of a solution $S(X)$ and suppose that w is not a block of one letter. Then there is a partition of this string into $u'w'^{k-2}v'$ such that

- $|w'| = |w|$ (and consequently $|u'| + |v'| = |u| + |v| + 2|w|$)
- the form of w' depends solely on w and does not depend on u, v (it does depend on the equation and on the order of blocks and pairs compressed by OneVarWordEq)
- the compression in one phase of OneVarWordEq compresses each w' from w'^{k-2} independently.

Proof. Let $w = a^\ell zb^r$, where $a, b \in \Sigma$, $\ell, r \geq 1$ and $z \in \Sigma^*$ does not start with a nor it ends with b. Let us represent uw^kv as $u(a^\ell zb^r)^kv$ and consider the first compression performed fully within the $zb^r(a^\ell zb^r)^{k-2}a^\ell z$, clearly some compression is made wholly within this blocks, see Fig. 1. Suppose for simplicity, that the first to be performed is a compression of pair, say cd. The case, when this first compression is a block compression is done similarly. We claim that all pairs cd that appeared within this fragment $zb^r(a^\ell zb^r)^{k-2}a^\ell z$ at the beginning are compressed at this moment. Assume for the sake of contradiction that this is not true. So this means that one of the letters, say c was already compressed in some other compression performed earlier. By the choice of the compressed pair, this c is compressed with a letter from outside of the fragment $zb(a^\ell zb^r)^{k-2}az$, there are two possibilities:

c is the last letter of $zb^r(a^\ell zb^r)^{k-2}a^\ell z$: Observe that the letter succeeding c is either b or a letter representing a compressed pair/string. In the latter case we do not make a further compression, so it is b. This is a contradiction: each c that is a last letter of z was initially followed by b, and so in fact this compression was performed earlier, contradiction with a choice of the compressed pair (as the first one fully within $zb^r(a^\ell zb^r)^{k-2}a^\ell z$).

c is the first letter of $zb^r(a^\ell zb^r)^{k-2}a^\ell z$: The argument is symmetric, with a preceding c in this case.

There are at least $k-1$ such pairs, each of them separated by $|w|-2$ letters, i.e. the $(cdz')^{k-2}cd$ is a substring of $zb^r(a^\ell zb^r)^{k-2}a^\ell z$, for some z' of length $|w|-2$, see Fig. 1. We take $w' = cdz'$ and let u' be the u concatenated with string proceeding the $(cdz')^{k-2}cd$ and v' the v concatenated with the string following this $(cdz')^{k-2}$ (note that it includes the ending cd). Clearly $|w'| = |w|$ and consequently $|u'|+|v'| = |u|+|v|+2|w|$. Note that each w' begins with cd, which is the first substring even partially within w' that is compressed, furthermore, each of those w' is also followed by cd. So the compression inside w' is done independently (because by the choice of cd there was no prior compression applied in w'). ☐

Consider a small solution, i.e. $S(X) = (w)^kv$, where w, v are almost periodic with period size N and side size $6N$. We claim that after each phase of OneVarWordEq, the solution will be of more general form

Claim 3. Suppose that a solution $S(X)$ is small. Then during the OneVarWordEq $S'(X) = u'(w')^{k'}v'$, where

- u' is a word
- w' is almost periodic with its period size at most $\frac{3}{4}$ of the period size (side size, repspectively) from the previous phase
- v' is a concatenation of almost periodic words, the sum of their period-sizes is at most $3N$ while the sum of their side-sizes plus the length of u' is at most $24N$

Proof. Clearly those conditions are met in the beginning.

So consider uw^kv at the beginning of one phase in the general case, in which we can apply Claim 2 to each w in uw^kv. Let $w = w_1w_2^\ell w_3$. Then by this claim we can represent $w_1w_2^\ell w_3$ as $w_1'w_2'^{\ell-2}w_3'$, where $|w_2'| = |w_2|$ and compression of each w_2' is independent (also, $|w_1'| + |w_3'| = |w_1| + |w_3| + 2|w_2|$). Consider then uw^kv which can be represented as

$$u\left(w_1'w_2'^{\ell-2}w_3'\right)^k v = uw_1'\left(w_2'^{\ell-2}w_3'w_1'\right)^{k-1}w_2'^{\ell-2}w_3'v$$

Observe that each $w_2'^{\ell-2}w_3'w_1'$ is delimited by w_2' (it includes it in the left end and to the right there is a copy of it which is not inside this $w_2'^{\ell-2}w_3'w_1'$) and each w_2' is compressed independently, so also each $w_2'^{\ell-2}w_3'w_1'$ is compressed independently, so in particular it is compressed in the same way. Thus uw^kv is compressed into $u'w'^{k-1}v'$, let us estimate their sizes.

The $\left(w_2'^{\ell-2}w_3'w_1'\right)^{k-1}$ is going to be compressed to some $\left(w_2''^{\ell-2}w_3''w_1''\right)^{k-1}$, where $|w_2''| \leq \frac{3}{4}|w_2'|$ and $|w_1''w_3''| \leq \frac{3}{4}|w_1'w_3'|$, hence both the period-size and side-size reduce by 1/4.

Concerning u', it is obtained as a compression of uw_1', so it has length at most $\frac{3}{4}|uw_1'|$. Concerning v', note that after prepending $w_2'^{\ell-2}w_3'v$ it is again a concatenation of periodical words, however, the sum of period-size is increased to $4N$ and the side-size increased by $|w_3'|$. Now, using Claim 2 those almost-periodical word can be refactored, so that the compression within each of the periodic part is independent (note that $w_2'^{\ell-2}$ does not need to be refactored). In this way the periodic-size is unchanged and the side-size increases by at most $6N$ (i.e. twice the period-size). Afterwards, this is all compressed and so the lengths reduce by at least one fourth. So the new period size is at most

$$\frac{3}{4}(N + 3N) = 3N \ ,$$

as claimed, while $|u'|$ plus the sum of side-sizes of v' is at most

$$\begin{aligned}
&\frac{3}{4}|uw_1'| + \frac{3}{4}\left(|w_3'| + \text{“original side size of } v\text{”} + \underbrace{6N}_{\text{additional side-size from the refactoring}}\right)\\
&= \frac{3}{4}\left((|w_1'| + |w_3'|) + (|u| + \text{“original side size of } v\text{”}) + 6N\right)\\
&= \frac{3}{4}\left(2N + 24N + 6N\right)\\
&= 24N
\end{aligned}$$

as claimed.

We are left to consider the border cases, in which w_2 is either a (non-trivial) block of letters or a single letter. Consider then the case in which w_2 is a block of a single letter. Then during the block compression w_2 is replaced with a single letter and the rest of the proof follows in a similar way. In particular, the claim holds in this case. When w_2 is a single letter the whole w_2^ℓ is replaced with a single letter, which we can include in w_1' or w_3' with $w_2' = \epsilon$, then the claim holds as well. ☐

Using Claim 3 we obtain that after $\mathcal{O}(1)$ phases the $S(X) = w^\ell v$ is reduced to $S'(X) = u'v'$, where v' is a concatenation of almost periodic words such that the sum of their period-size is at most $3N$ and the sum of their side size plus $|u'|$ is at most $24N$. Applying iteratively to them Claim 2 we obtain that after $\mathcal{O}(1)$ phases this is replaced with a single letter. ☐

4.4. Storing of an equation.

While the long words are stored exactly as they used to, the short words are stored more efficiently: we keep a table of short words and the pointer from the word-table points to the table of short words, in this way all identical short words are stored only once. We say that such a representation is *succinct* and its size is the sum of lengths of words stored in it. Note that we do *not* include the size of the word table. Additionally, all long words are kept on a doubly linked list (recall that each of them stores whether it is the first or the last word). Note that in this way we do not need to actually read the whole equation in order to compress it: it is enough to read the words in the succinct representation.

We first show that such a storage makes sense, i.e. that if two short words become equal, they remain equal in the following phases.

Lemma 14. *Consider any words A and B in the input equation. Suppose that during* OneVarWordEq *they were transformed to $A' = B'$, none of which is a first or last word in one of the equations. Then $A = B$ if and only if $A' = B'$.*

Proof. By induction on operation performed by OneVarWordEq. Since none of the A', B' is the first or last word in the equation, it means that during the whole OneVarWordEq they had X to the left and to the right. So whenever a letter was left-popped or right-popped from X, it was prepended or appended to both A and B; the same applies to cutting prefixes and suffixes. Compression is never applied to a crossing pair or a crossing block, so after it two strings are equal if and only if they were before the operation. The removal of letters is applied only to first and last words, so it does not apply to words considered here. Partitioning the equation into subequations does not affect the equality of words. ☐

The compression (both pair and block) can be performed on succinct representation in linear time.

Lemma 15. *The compression in one phase of* OneVarWordEq *can be performed in time linear in size of the succinct representation.*

Proof. The long words are stored in a list and we can compress them without the need of reading the word table. We know which one of them is first or last, so when letters are popped from X we know what letters are appended/prepended to each of those words. Since they are stored explicitly, the claim for them follows from the analysis of the original version of OneVarWordEq.

For the short words stored in the tables, from Lemma 14 it follows that if an explicit word A appears twice in the equations (both times not as a first, nor last word of the equation) it is changed during OneVarWordEq in the same way at both those instances. So it is enough to perform the operations on the words stored in the tables, doing so as in the original version of OneVarWordEq takes time linear in the size of the tables of short words. ☐

The words stored in the tables are of two types: normal and overdue. The *normal* words are substrings of the long words or A_0^2 and consequently the sum of their sizes is proportional to the size of the long words. A word becomes *overdue* if at the beginning of the phase it is not a substring of a long word or A_0^2. It might be that it becomes a substring of such a word later, it does not stop to be an overdue word in such a case. The new overdue words can be identified in linear time: this is done by constructing a suffix array for a concatenation of long and short strings appearing in the equations.

Lemma 16. *In time proportional to the sum of sizes of the long words plus the number of overdue words we can identify the new overdue words.*

Furthermore in the same time bounds we can return for each overdue word a sorted list of its appearances.

Proof. Consider all long words $A_0, \ldots, A_m$ (with or without multiplicities, it does not matter) and all short (not already overdue) words $A'_1, \ldots A'_{m'}$, without multiplicities; in both cases this is just a listing of words stored in the representation (except for old overdue words). We construct a suffix array for the string

$$A_0^2\$A_1\$\ldots A_m\$A'_1\$\ldots A'_{m'}\#,$$

which can be done in linear time [9].

Now A'_i is a factor in some A_j (the case of A_0^2 is similar, it is omitted to streamline the presentation) if and only if for some suffix A''_j of A_j the strings $A''_j\$A_{j+1}\ldots A_m\$A'_1\$\ldots\$A'_{m'}\#$ and $A'_i\$\ldots\$A'_{m'}\#$ have a common prefix of length at least $|A'_i|$. In terms of a suffix array, the entries for $A'_i\$\ldots\$A'_{m'}\#$ and $A''_j\$A_{j+1}\ldots\$A_m\$A'_1\$\ldots\$A'_{m'}\#$ should have a common prefix of length at least $|A'_i|$. Recall that the length of the longest common prefix of two suffixes stored at positions $p < p'$ in the suffix array is the minimum of $LCP[p]$, $LCP[p+1]$, $\ldots$, $LCP[p']$. For fixed suffix $A'_i\$\ldots\$A'_{m'}\#$ consider the $A''_j\$A_{j+1}\ldots\$A_m\$A'_1\$\ldots\$A'_{m'}\#$ with which it has the longest common prefix. Then it is either the

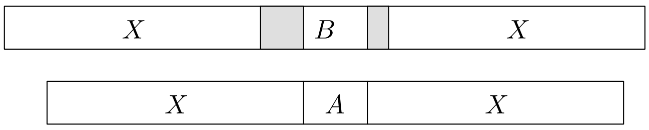

FIGURE 2. A is arranged against B. The periods of length at most $|B| - |A|$ are in ligther grey. Since $A \neq B$, at least one of the is non-empty.

first previous or first next suffix of this form in the suffix array. Thus the appropriate computation can be done in linear time: we first go down in the suffix array, storing the last spotted entry corresponding to a suffix of some long A_j, calculating the LCP with consecutive suffixes and storing them for the suffixes of the form $A'_i\$\dots\$A'_{m'}\#$. We then do the same going from the bottom of the suffix array. Lastly, we choose the larger from two stored values; for $A'_i\$\dots\$A'_{m'}\#$ it is smaller than $|A'_i|$ if and only if A'_i just became an overdue word.

Concerning the running time, the size of the string is proportional to the length of the long words plus the length of the words that are still normal, i.e. not overdue (and this is proportional to the length of the long words) plus the size of the new overdue words. As each of the latter is of size at most N, the sum of their sizes is proportional to their number. ☐

The main property of the overdue words is that they can be removed from the equations in $\mathcal{O}(1)$ phases after becoming overdue. This is shown by a serious of lemmata.

First we need to define what does it mean that for solution word A in one side of the equation is at the same position as its copy on the other side of the equation: we say that for a substitution S the A_i (or its subword) is *arranged against* B_j ($S(X)$ for some fixed appearance of X) if the position within $S(\mathcal{A})$ occupied by this (subword of) A_i are within the positions occupied by B_j ($S(X)$, respectively).

Lemma 17. *Consider a word A in a phase in which it becomes overdue. Then for each solution $S(X)$ either S is small or in every $S(\mathcal{A}_i) = S(\mathcal{B}_i)$ each word A is arranged against another instance of A.*

Proof. Consider an equation and a solution S such that in some $S(\mathcal{A}_i) = S(\mathcal{B}_i)$ a factor coming from an overdue word A is not arranged against a factor coming from an overdue word A. There are several cases:

A is arranged against $S(X)$. Note that in this case A is a substring of $S(X)$. Either $S(X)$ is a substring of A_0 or $S(X) = A_0^k A'_0$, where A'_0 is a prefix of A_0. In the former case A is a factor of A_0, which is a contradiction, in the latter it is a factor of A_0^{k+1}. As A_0 is long and A short, it follows that $|A| < |A_0|$ and so A is a factor of A_0^2, contradiction with the assumption that A is overdue.

A is arranged against some word. Since A is an overdue word, this means that it is arranged against short word B. Note that both A and B are preceded and succeeded by $S(X)$, since $A \neq B$ we conclude that $S(X)$ has a period at most $|B| - |A|$, see Fig. 2; in particular S is small.

Other case. Some part of A is arranged against $S(X)$ and as A is preceded and succeeded by $S(X)$, which means that $S(X)$ has a period at most $|A|$, similarly as in the previous case; so in particular S is small. ☐

Observe that due to Lemmata 13 and 17 the overdue words can be removed in $\mathcal{O}(1)$ phases after their introduction: suppose that A becomes an overdue word in phase ℓ. Any solution, in which an overdue word A is not arranged against another copy of A is small and so it is reported after $\mathcal{O}(1)$ phases. In the following phase an equation $\mathcal{A}'_i XAX\mathcal{A}''_i = \mathcal{B}'_i XAX\mathcal{B}''_i$, where $\mathcal{A}_i$ and $\mathcal{B}_i$ do not have A as a word, is equivalent to two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ and this procedure can be applied recursively to $\mathcal{A}''_i = \mathcal{B}''_i$. In this way, all appearances of A are removed and no solutions are lost in the process. There may be many overdue strings so the process is a little more complicated, however, it can be implemented with ease in time proportional to the size of the removed overdue words and so this time summed over all phases is $\mathcal{O}(n)$.

Lemma 18. *Consider the set of overdue words introduced in phase ℓ. Then in phase $\ell + c$ (for some constant c) we can remove all those words from the equations. The obtained set of equations has the same set of solutions. The amortised time spend on removal of overdue words, over the whole run of* OneVarWordEq, *is $\mathcal{O}(\#_X)$.*

Proof. Consider any word A that become overdue in phase ℓ and any solution S of this equation, such that in some $S(\mathcal{A}_i) = S(\mathcal{B}_i)$ the explicit word A is not arranged against another instance of the same explicit word. Then due to Lemma 17 the $S(X)$ is small. Consequently, from Lemma 13 this solution is reported before phase $\ell + c$, for some constant c. So any solution S' in phase $\ell + c$ corresponds to a solution S from phase ℓ that had each explicit A arranged in each $S(\mathcal{A}_i) = S(\mathcal{B}_i)$ against another instance of the explicit A. Since all operations in a phase either transform solution, implement the pair compression of implement the blocks compression for a solution $S(X)$, it follows that in phase $\ell + c$ the corresponding overdue words A' are arranged against each other in $S'(\mathcal{A}'_i) = S'(\mathcal{B}'_i)$.

This observation allows removing all overdue words introduced in phase ℓ. Let C_1, C_2, $\ldots$, C_m (in phase $\ell + c$) correspond to all overdue words introduced in phase ℓ. For each overdue word A the list of pointers to its appearances in left-hand sides of the equations and right-hand sides of the equations are sorted according to the order of their appearances, see Lemma 16. We go through those lists, if the first appearances of A in the left-hand sides and right-hand sides are in different equations then the equations are not satisfiable, as this would contradict that in each solution both A is arranged against its copy. Otherwise, they are in the same equation $\mathcal{A}_i = \mathcal{B}_i$, which is of the form $\mathcal{A}'_i XAX\mathcal{A}''_i = \mathcal{B}'_i XAX\mathcal{B}''_i$, where $\mathcal{A}'_i$ and $\mathcal{B}'_i$ do not have any appearance of A within them. We split it into two equations $\mathcal{A}'_i = \mathcal{B}'_i$ and $\mathcal{A}''_i = \mathcal{B}''_i$ and we trim them so that they are in the form described in (1). Note that as new equations are created, we need to reorganise the pointers from the first/last words in the equations, however, this is easily done in $\mathcal{O}(1)$ time. The overall cost can be charge to the removed X, which makes in total at most $\mathcal{O}(\#_X)$ cost. □

Lemma 19. *The running time of* OneVarWordEq*, except for time used to test the solutions, is* $\mathcal{O}(n)$*.*

Proof. By Lemma 15 the cost of compression is linear in terms of the size of the succinct representation. When the overdue words are excluded, this size is proportional to the total length of long words. Since by Lemma 8 this sum of lengths decreases by a constant in each phase, the sum of those costs is linear in n.

Consider the cost related to the overdue word. Note that a word is overdue for only $\mathcal{O}(1)$ phases, see Lemma 18. So the compression time for them is only $\mathcal{O}(1)$ per word, which sums to $\mathcal{O}(n)$ where all words are included. It was already shown in Lemma 16 that overdue words can be identified in time proportional to the sum of lengths of long words plus the number of new overdue words, so this cost is $\mathcal{O}(n)$, when summed over all phases. Lastly, Lemma 18 shows that also the time spend on removing those words from the equations is $\mathcal{O}(n)$. □

4.5. **Testing.**

We already know that storing and compressing the equations can be performed in linear time, it still remains to explain how to test the solutions fast, i.e. how to perform TestSimpleSolution when all first and last words are still long (note that TestSolution is performed once, so it can take time proportional to n). Recall that TestSimpleSolution checks whether S is a solution by comparing $S(\mathcal{A}_i)$ and $S(\mathcal{B}_i)$ letter by letter, replacing X with a^ℓ on the fly. We say that in such a case a letter b in $S(\mathcal{A}_i)$ is *tested against* the corresponding letter in $S(\mathcal{B}_i)$. Consider two letters, from A_i and B_j, that are tested against each other. If one of A_i and B_j is long, this can be amortised against the length of the long word. The same applies when one of the words A_{i-1}, A_{i+1}, B_{j-1} or B_{j+1} is long. So the only problematic case is when all of those words are short. To deal with this case efficiently we distinguish between different test types, in which we exploit different properties of the solutions to speed up the tests.

4.5.1. *Test types.*

Suppose that for a substitution S a letter from A_i is tested against a letter from $S(XB_j)$ (note that there is some asymmetry regarding $\mathcal{A}$-words and $\mathcal{B}$-words in the definition, this is a technical detail without an importance). We say that this test is:

- **protected:** if at least one of A_i, A_{i+1}, B_i, B_{i+1} is long
- **failed:** if A_i, A_{i+1}, B_j and B_{j+1} are short and a mismathch for S is found till the end of A_{i+1} or B_{j+1};
- **aligned:** if $A_i = B_j$ and $A_{i+1} = B_{j+1}$, all of them are short and the first letter of A_i is tested against the first letter of B_j;
- **misaligned:** if all of A_i, A_{i+1}, B_j, B_{j+1} are short, $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ and this is not an aligned test;
- **periodical:** if $A_{i+1} = A_i$, $B_{j+1} = B_j$, all of them are short and this is not an aligned test.

It is easy to show that there are no other tests, see Lemma 20. We separately calculate the cost of each type of tests. For failed tests note that they take constant time per phase and we know that there are $\mathcal{O}(\log n)$ phases. For protected tests, we charge the cost of the protected test to the long word and only $\mathcal{O}(|C|)$ such tests can be charged to one long word C in a phase. On the other hand, each long word is shortened by a constant factor in a phase and so this cost can be charged to those removed letters and thus the total cost of those tests (over the whole run of OneVarWordEq) is $\mathcal{O}(n)$.

In case of the misaligned tests, it can be shown that S in this case is small and that it is tested at the latest $\mathcal{O}(1)$ phases after the last of A_{i+1}, A_i, B_{i+1}, B_i becomes short, so this cost can be charged to, say, B_i becoming short and only $\mathcal{O}(1)$ such tests are charged to this B_i (over the whole run of the algorithm). Hence the total time of such tests is $\mathcal{O}(n)$.

For the aligned tests, consider the consecutive aligned tests, they correspond to comparison of $A_iXA_{i+1}\dots A_{i+k}X$ and $B_jXB_{j+1}\dots B_{j+k}X$, where $A_{i+\ell} = B_{j+\ell}$ for $\ell = 1,\dots,k$. Then it can be shown that the previous test is either misaligned or protected, so if the cost of all those tests can be bounded by $\mathcal{O}(1)$, they can be associated with the succeeding test. Note that instead of performing the aligned tests, it is enough to identify the maximal (syntactically) equal substrings of the equation and from Lemma 14 it follows that this corresponds to the (syntactical) equality of substrings in the original equation. Such an equality can be tested in $\mathcal{O}(1)$ using a suffix array constructed for the input equation (and general LCP queries on it).

For the periodical test suppose that we are to test the equality of (suffix of) $S((A_iX)^\ell)$ and (prefix of) $S(X(B_jX)^k)$. If $|A_i| = |B_j|$ then the test for A_{i+1} and B_{j+1} is the same as for A_i and B_j and so can be skipped. If $|A_i| > |B_j|$ then the common part of $S((A_iX)^\ell)$ and $S(X(B_jX)^k)$ have periods $|S(A_iX)|$ and $|S(B_jX)|$ and consequently has a period $|A_i| - |B_j| \leq N$. So it is enough to test first common $|A_i| - |B_j|$ letters and check whether $|S(A_iX)|$ and $|S(B_jX)|$ have period $|A_i| - |B_j|$.

This yields that the total time of testing is linear. The details are given in the next subsections.

Lemma 20. *Each test is either failed, protected, misaligned, aligned or periodical.*

Proof. Consider a test of a letter from A_i and $S(XB_j)$. If any of A_{i+1}, B_{j+1}, A_i or B_j is long then it is protected. So consider the case in which all of them are short. it might be that this test is failed. Otherwise, if the first letter of A_i and B_j are tested against each other and $A_i = B_j$ and $A_{i+1} = B_{j+1}$ then the test is aligned. Otherwise, if $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$ then it is misaligned. In the remaining case $A_{i+1} = A_i$ and $B_{j+1} = B_j$, so this is a periodical test. □

Note that when calculating the time spend on testing, we can consider only the one in which at least one letter is from a word: the tests in which both letters are from $S(X)$ take only constant time per test (as we compare the suffix of a^k with a^k we can move forward the whole suffix) and so we can charge it to the test of a neighbouring word.

4.5.2. *Failed tests.* There are $\mathcal{O}(1)$ substitutions tested per phase and so

Lemma 21. *The number of all failed tests is $\mathcal{O}(\log n)$ over the whole run of* OneVarWordEq.

Proof. As noticed, there are $\mathcal{O}(1)$ substitutions tested per phase. Note that letters from at most two short blocks in some $\mathcal{A}_i$ and two short blocks in some $\mathcal{B}_i$ take part in the failed tests. Since short blocks have length at most N, we conclude that there are $\mathcal{O}(1)$ failed tests per phase and so $\mathcal{O}(\log n)$ failed tests in total, as there are $\mathcal{O}(\log n)$ phases. □

4.5.3. *Protected tests.* As already claimed, the total number of protected tests is linear in terms of length of long words.

Lemma 22. *In one phase the total number of protected tests is proportional to the length of the long words. In particular, there are $\mathcal{O}(n)$ such test during the whole run of* OneVarWordEq.

Proof. Suppose that a letter from A_i takes part in the protected test, then one of A_i, A_{i+1}, B_j, B_{j+1} is long, we charge the cost of this test to this long word (any of them, if there is a choice). Fix some long word $A_{i'}$. It can be charged with cost of tests of its own letter ($\mathcal{O}(|A_{i'}|)$, as well as protected tests of letters from $A_{i'-1}$ and $A_{i'+1}$ (when they are short), so $\mathcal{O}(1)$ tests. Furthermore, let the first letter of $A_{i'}$ under the tested substitution is against a letter from $B_{j'}$ and the last against a letter from $B_{j''}$. Then also the tests from (short among) $B_{j'-1}$, $B_{j'}$, $B_{j''}$ and $B_{j''+1}$ can be charged against $A_{i'}$, but again there are only $\mathcal{O}(1)$ of them. So in total $A_{i'}$ is charged only $\mathcal{O}(|A_{i'}|)$ in a phase.

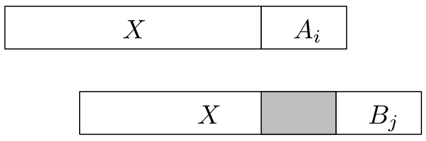

Figure 3. A letter from B_j is arranged against the letter from A_i. The period of $S(X)$ is in grey.

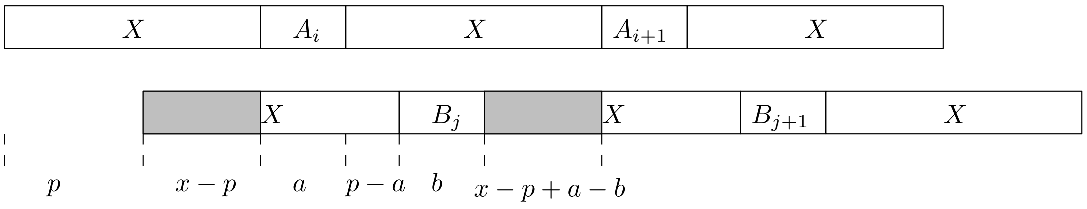

Figure 4. The letters of B_j are arranged against the letters from $S(X)$. The lengths of fragments of text are beneath the picture, between slashed lines. Comparing the positions of the first and second $S(X)$ yields that p is a period, second and third that $x - p + a$. The corresponding borders of $S(X)$ are marked in grey.

From Lemma 8 the sum of length of long words drops by a constant factor in each phase, and as in the input it is at most n, the total sum of the cost of protected test is $\mathcal{O}(n)$. ☐

4.5.4. *Misaligned tests.* We first generalise the notion of a misaligned test to a more general setting, in which S is an arbitrary substitution and not a tested one. We say that A_i and B_j that are blocks from two sides of one equations $\mathcal{A}_\ell = \mathcal{B}_\ell$ are *misaligned for a substitution* S if

- a mismatch for S is found till the end of A_{i+1} or B_{j+1};
- all A_{i+1}, A_i, B_{j+1} and B_j are short;
- they are not aligned, i.e. it does not hold that $A_i = B_j$ and $A_{i+1} = B_{j+1}$ and the first letter of A_i is at the same position as the first letter of B_j under substitution S;
- the position of the first letter of A_i in $S(\mathcal{A}_\ell)$ is among the position of $S(XB_j)$ in $S(\mathcal{B}_\ell)$.

Lemma 23. *When the A_i and B_j are misaligned for a solution S then S is small.*

Proof. Suppose that A_i and B_j are from an equation $\mathcal{A}_\ell = \mathcal{B}_\ell$. In the proof we consider only one of the symmetric cases, in which A_i begins not later than B_j, the remaining case is shown in the same way.

By definition, each of A_{i+1}, A_i, B_{j+1} and B_j is short and consequently is not the first word in $\mathcal{A}_\ell = \mathcal{B}_\ell$. So consider how $S(XA_iXA_{i+1}X)$ and $S(XB_jXB_{j+1}X)$ are arranged against each other in $\mathcal{A}_\ell = \mathcal{B}_\ell$. If any letter of B_j is arranged against a letter of A_i and $A_i \neq B_j$ then $S(X)$ is periodic, with period smaller than $|A_i| \leq N$, see Fig. 3. If $B_j = A_i$ and their first letters are against each other then as $A_{i+1} \neq B_{j+1}$ and their first letters are also arranged against each other then we can make the same analysis as in the previous case, this time for A_{i+1} and B_{j+1}. In each of those cases S is small. The same argument can be applied to the letters of B_{j+1} (note that when $B_{j+1} = A_{i+1}$ and their first letters are aligned we look at A_i and B_j).

So, in the following we may assume that letters of B_j (B_{j+1}) are arranged against the letters from $S(X)$. Let $a = |A_i|$, $b = |B_j|$ and $x = |S(X)|$, as in Fig. 4. Consider first the (main) case, in which $a \neq b$, we consider the case in which $a > b$, the other is similar. Let p denote the offset between the $S(X)$ preceding A_i and the one proceeding B_j, see Fig. 4. Then $S(X)$ has a borders of length $x - p$ and $x - p + a - b$, see Fig. 4. Then the shorter border (of length $x - p$) is also a border of the longer one (length $x - p + a - b$), hence the border of length $x - p + a - b$ has a period $a - b$. Thus the prefix of $S(X)$ of length $x - p$ is of the for $w^k u$, where $|w| = a - b$ and $|u| \leq a - b$. Then the prefix of $S(X)$ of length $x - p + a$ (consider the second $S(X)$ on Fig. 4) is of the form $w^k u A_i$. But as $S(X)$ has a period $x - p + a$, the whole $S(X)$ is of the form $(w^k u A_i)^\ell w'$, where w' is a prefix of $w^k u A_i$, hence it is small.

So consider now the case, in which $|A_i| = |B_j|$. If $|A_{i+1}| \neq |A_i|$ then $|B_j| \neq |A_{i+1}|$ and we can repeat the same reasoning as above, with B_j and A_{i+1} taking the roles of A_i and B_j, which shows that S is small. So consider the case in which $|A_{i+1}| = |A_i|$. If $|B_j| \neq |B_{j+1}|$ we also obtain that S is small. So we are left with $|A_{i+1}| = |A_i|$ and $|B_j| = |B_{j+1}|$. Then A_{i+1} is arranged against the same letters in $S(X)$ as A_i and B_{j+1} is arranged against the same letters in $S(X)$ as B_j. Since $A_{i+1} \neq A_i$ or $B_{j+1} \neq B_j$, some of those tests fail, contradicting the assumption that S is a solution. □

Lemma 24. *There are $\mathcal{O}(n)$ misaligned test during the whole run of* OneVarWordEq.

Proof. We first show that if A_i and B_j are misaligned for S then they were (for a corresponding solution) already when the last among $\mathcal{A}$ $(i+1)$-word, $\mathcal{A}$ i-word, $\mathcal{B}$ $(j+1)$-word and $\mathcal{B}$ j-word became short. Note that it is enough to show the claim for the previous phase end then apply the induction.

Claim 4. Suppose that A_i and B_j are misaligned for S. If at a previous phase all A'_{i+1}, A'_i, B'_{j+1} and B'_j were short then A'_i and B'_j were misaligned for the corresponding S'.

Proof. First note that by Lemma 14 it holds that $A'_i \neq A'_{i+1}$ or $B'_j \neq B'_{j+1}$. As additionally all A'_{i+1}, A'_i, B'_{j+1} and B'_j are short, then the only reason why A'_i and B'_j could not be misaligned for S' is that $A'_i = B'_j$, $A'_{i+1} = B'_{j+1}$ and under S' the first letters of those A'_i and B'_j are arranged against each other. Note that left-popping and right popping letters from X does not change those equalities, nor the fact that the first letters of those words are arranged against each other. The same applies to pair compression and block compression. Hence, also $A_i = B_j$, $A_{i+1} = B_{j+1}$ and the first letters of A_i and B_j are arranged against each other, contradiction. □

Using Claim 4 we can give a bound on when a word can take part in a misaligned test for a solution S (not that this not necessarily apply to all substitutions).

Claim 5. Let the ℓ-th phase be the one in which the last of A'_{i+1}, A'_i, B'_{j+1} and B'_j became short. Then there is no misaligned test of letter from A_i against a letter from $S(XB_j)$ for a solution S in phase $\ell + c$ or later, for some constant c.

Proof. Suppose that A_i and B_j have a misaligned test for a solution S. By inductive argument on Claim 4 it follows that A'_i and B'_j are misaligned for the corresponding S'. By Lemma 23 it follows that S' is small and so it is tested at the latest in phase $\ell + c$ (for some constant c). □

So whenever (in phase ℓ') a letter from A_i has a misaligned test against a letter from $S(XB_j)$ we can check, in which turn ℓ the last among words A_i, A_{i+1}, B_j B_{j+1} became small. If $\ell + c < \ell'$ then we can terminate the test, as we know already that S is not a solution. This allows an estimation on the number of misaligned tests. Firstly, there are at most $\mathcal{O}(\log n)$ of tests that terminate the whole testing, the argument is similar as in the proof of Lemma 21. Otherwise, the cost of the test (of a letter from A_i tested against $S(XB_j)$) is charged to the last one among A_i, A_{i+1}, B_j, B_{j+1} that became short. For a fixed substitution, a fixed word can be charged only from a constant number of short words, so $\mathcal{O}(1)$ cost in total. There are only $\mathcal{O}(1)$ substitutions per phase and moreover the word is charged only for c phases after it became short, so in total there is $\mathcal{O}(1)$ charges to this word. Summing over all words in the instance yields the claim of the lemma. □

4.5.5. *Aligned tests.* Suppose that we test the first letter of some A_i against the first letter in B_j (both of those words are short), where $A_{i-1} \neq B_{j-1}$ or one of them was long (i.e. the tests for previous words were not aligned). We can preprocess (in $\mathcal{O}(n)$ time) the input equation (building a suffix array equipped with a structure answering general LCP queries) so that in $\mathcal{O}(1)$ time it returns the smallest $k \geq 0$ such that $A_{i+k} \neq B_{j+k}$. In this way we perform all equality tests for $A_i X A_{i+1} X \dots A_{i+k-1} X = B_j X B_{j+1} X \dots X B_{j+k-1} X$ in $\mathcal{O}(1)$ time.

Lemma 25. *In $\mathcal{O}(n)$ we can build a data structure such that given two A_i and B_j it in $\mathcal{O}(1)$ time returns the smallest $k \geq 0$ such that $A_{i+k} \neq B_{j+k}$.*

Proof. Let A'_i, B'_j etc. denote the corresponding original words of the input equation. Observe that by Lemma 14 $A'_{i+\ell} = B'_{j+\ell}$ if and only if $A_{i+\ell} = B_{j+\ell}$. Hence, it is enough to be able to answer such queries for the input equation.

To this end we build a suffix array [9] for the input equation, i.e. for $A_1 X A_2 X \dots A_{n_\mathcal{A}} X B_1 X B_2 X \dots B_{n_\mathcal{B}} \$$. Now, the lcp query for suffixes $A_i \dots \$$ and $B_j \dots \$$ returns the length of the longest common prefix. What we want is the information, is the number of words in the common prefix, which corresponds to the number of Xs in this common substring. And this information can be easily preprocessed and stored in the suffix array. For instance, we can for each position ℓ in $A_1 X A_2 X \dots A_{n_\mathcal{A}} X B_1 X B_2 X \dots B_{n_\mathcal{B}} \$$

store, how many Xs are before it in the string and store this in the table *prefX*. Then when for a suffixes starting a position p we get that its common prefix is of length k, we return $\mathit{prefX}[p+k]-\mathit{prefX}[p]$, which is the number of Xs in the common prefix in such a case. □

Using this data structure performing all aligned tests is easy: whenever we test the first letter of A_i against the first letter of some B_j, we use this structure, obtain k and jump to the test of the first letter of A_{i+k} with the first letter of B_{j+k}. If $k = 0$ then this is not the aligned test and we add the cost of using the data structure to the cost of the test. If $k > 0$ then $A_{i+k} \neq B_{j+k}$ and $A_{i+k-1} = B_{j+k-1}$. Thus the tests for letters of A_{i+k-1} are either protected, misaligned or failed (they cannot be periodical, as $A_{i+k-1} = B_{j+k-1}$ and $A_{i+k} \neq B_{j+k}$ implies that $A_{i+k} \neq A_{i+k-1}$ or $B_{j+k-1} \neq B_{j+k}$). So we can associate the cost of those aligned tests with the cost of a protected, misaligned or failed test, and so the time spend on them is at most linear over the whole run of OneVarWordEq.

Lemma 26. *The total cost aligned test and the usage of the suffix array is $\mathcal{O}(n)$.*

Proof. We formalise the discussion above. When the first letter of A_i and B_j should be tested against each other and $A_i \neq B_j$ then this is not an aligned test (the cost of checking whether $A_i = B_j$ is then added to the cost of the following letter-test). Otherwise this is an aligned test, we add the cost of checking whether $A_i = B_j$ to the cost of the aligned test.

In $\mathcal{O}(1)$ we get the smallest k such that $A_{i+k} \neq B_{j+k}$ or one of them is in the other equation. We then jump straight to the test of A_{i+k} or B_{j+k} (those are different tests if one of them is a first word in an equation). Consider A_{i+k-1} and B_{j+k-1}. There are the following cases:

one of A_{i+k-1} or B_{j+k-1} is a last word: which shows that it is long, contradiction

$A_{i+k-1}X$ or B_{j+k-1X} ends one side of the equation: Then the other side is ended differently (they cannot both end in X) and so the test for one of A_{i+k-1} or B_{j+k-1} is failed

both A_{i+k-1} and B_{j+k-1} are within the same equation: we show that in this case the test for the first letters of A_{i+k-1} and B_{j+k-1} is protected or failed or misaligned. Firstly, since $A_{i+k-1} = B_{j+k-1}$ and $A_{i+k} \neq B_{j+k}$ then $A_{i+k-1} \neq A_{i+k}$ or $B_{j+k-1} \neq B_{j+k}$. Hence the test for the first letters of A_{i+k-1} and B_{j+k-1} cannot be aligned nor periodical. So it is misaligned, protected or failed.

Hence we can associate the $\mathcal{O}(1)$ cost of all aligned tests for $A_iX \dots A_{i+k-1}X$ and $B_JX \dots B_{j+k-1}X$, with either a failed, protected or misaligned test. Thus in total they take $\mathcal{O}(n)$ time. □

4.5.6. *Periodical tests.* In order to identify the periodical tests we keep for each short A_i the value k such that A_{i+k} is the first word that is different from A_i or in another equation, those are easy to calculate at the beginning of each phase. Now when a periodical test is done, i.e. we test letters from a suffix of $S((AX)^k)$ against the letters from (prefix of) $S((BX)^\ell)$ then we use that $S(BX)^\ell$ has a period $|S(BX)|$, hence it is enough to perform this test manually for $S((AX)^2)$ and $S((BX)^2)$ and then check whether $S((AX)^k)$ has a period $|S(BX)|$. All of this can be done in time $\mathcal{O}(|A| + |B|)$, so constant. Furthermore, the next test is either protected, misaligned or failed, which yields that the number of periodical tests is $\mathcal{O}(n)$.

Lemma 27. *Performing all periodical tests and the required preprocessing takes in total $\mathcal{O}(n)$ time*

Proof. We first show how to calculate for each short A_i (and B_j) the k such that A_{i+k} is the first word that is different from A_i or it is in another equation.

At the end of the phase we can list all words A_i that become short in this phase, ordered from the left to the right. Note that this takes at most the time proportional to the length of all long words from the beginning of the phase, so $\mathcal{O}(n)$ in total. Consider any A_i on this list, note that

- if $A_{i+1} \neq A_i$ or it is in the next equation, then A_i should store $k = 1$;
- if $A_i = A_{i+1}$ and they are in the same equation then A_{i+1} also became short in this phase and so it is on the list and consequently A_i should store 1 more than A_{i+1}.

So we read the list from the right to the left, let A_i be an element on this list. Using the above condition, we can establish in constant time the value stored by A_i. This operation is performed once per block, so in total takes $\mathcal{O}(n)$ time.

So suppose that during the tests, when the first letter of A_i is tested against the letter of $S(XB_j)$, the k_A and k_B stored by A_i and B_j are both greater than 1. Consider the one of A_{i+k_A} and B_{j+k_B} which ends first under substitution S (this can be determined in $\mathcal{O}(1)$ by simply comparing the lengths), without loss of generality let it be the latter. There are the following cases:

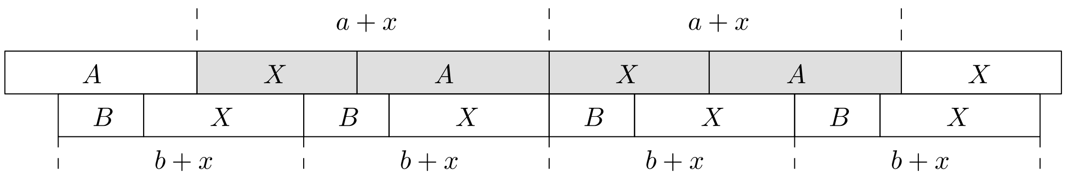

FIGURE 5. The case of $a > b$. The part of $S((XA_i)^2)$ that has a period $a+x$ and $b+x$ is in grey.

$B_{j+k_B} \neq B_{j+k_B-1}$**:** Then the test for B_{j+k_B-1} is not periodical. Furthermore, it cannot be aligned: if it were, then B_{j+k_B-1} is aligned against A_{i+k} for some $k < k_A$ and so $B_{j+k_B-1} = A_{i+k}$ implies that $A_i = B_j$ and they are aligned, contradiction. So it is either misaligned, protected or failed.

$B_{j+k_B}X$ **ends the equation:** in this case either the other side of the equation does not end at this point, so the test for B_{j+k_B-1} is failed, or A_{i+k} for $k \leq k_A$ is the last word, so the test for the letters in A_{i+k_A} are protected.

B_{j+k_B} **is the last word:** then the test for B_{j+k_B-1} is failed, as we consider the case in which under the substitution S the A_{i+k_A} ends later than B_{j+k+B}.

So either the test for B_{j+k_B} is either protected, failed , aligned or misaligned or the test for A_{i+k_A} is protected. Since there are in total $\mathcal{O}(n)$ of such tests it is enough to show that the test for the common part of $S(A_iX \cdots XA_{i+k_A-1}X)$ and $S(B_jX \cdots XB_{j+k_B-1}X)$ can be performed in $\mathcal{O}(1)$ time, which is associated with the cost of the following test.

Let $a = |A_i|$, $b = |B_j|$ and $x = |S(X)|$. First consider the simpler case in which $a = b$. Let $k = \min(k_A, k_B)$. Then the tests for $A_{i+1}, \ldots, A_{i+k-1}$ are identical as for A_i, and so it is enough to perform just the test for A_i and then jump right to A_{i+k}.

So let us now consider the case in which $a > b$ and that the first letter of A_i under the substitution S begins earlier than the first letter of B_j, the other are done in the same way. Observe that when the whole $S((B_jX)^\ell)$ is within $S((A_iX)^3)$ then this can be tested in constant time in a naive way: the length of $S((A_iX^3))$ is $3(a+x)$ while the length of $S(B_jX)^\ell$ is $\ell(b+x)$. Since a/b is at most N we obtain that $\ell \leq 3N$ and so the whole test can be done in $\mathcal{O}(1)$.

So consider the remaining case, see Fig. 5 for an illustration. First $S(XA_iXA_i)$ has period $x+a$. However, it is covered with $S((B_jX)^\ell)$, so it also has period $x+b$. Since $x+a+x+b \leq 2x+2a$, it follows that also the $p = \gcd(x+a, x+b) \leq a-b$ is a period of $S(XA_iXA_i)$ and consequently, of $S(A_iX)$ and $S(B_jX)$. So, to perform the test for the common part of $S(A_iX \cdots XA_{i+k_A-1}X)$ and $S(B_jX \cdots XB_{j+k_B-1}X)$ it is enough to calculate p, test whether $S(A_iX)$, $S(B_jX)$ have period p and then compare their first common p letters. All of this can be done in $\mathcal{O}(1)$, since $p \leq a-b \leq N$ (note also that calculating p can be done in $\mathcal{O}(1)$, as $\gcd(x+a, x+b) = \gcd(a-b, x+b)$ and $a-b \leq N$). ☐

Open problems. Is it possible to remove the usage of range minimum queries from the algorithm without increasing the running time? Can the recompression approach be used to speed up the algorithms for the two variable word equations?

Acknowledgements. I would like to thank A. Okhotin for his remarks about ingenuity of Plandowski's result, which somehow stayed in my memory; P. Gawrychowski for initiating my interest in compressed membership problems and compressed pattern matching, exploring which eventually led to this work as well as for pointing to relevant literature [12, 14]; J. Karhumäki, for his explicit question, whether the techniques of local recompression can be applied to the word equations; last not least, W. Plandowski for his numerous comments and suggestions on the recompression applied to word equations.

REFERENCES

[1] Omer Berkman and Uzi Vishkin. Recursive star-tree parallel data structure. *SIAM J. Comput.*, 22(2):221–242, 1993.

[2] Witold Charatonik and Leszek Pacholski. Word equations with two variables. In Habib Abdulrab and Jean-Pierre Pécuchet, editors, *IWWERT*, volume 677 of *Lecture Notes in Computer Science*, pages 43–56. Springer, 1991.

[3] Robert Dąbrowski and Wojciech Plandowski. Solving two-variable word equations. In Josep Díaz, Juhani Karhumäki, Arto Lepistö, and Donald Sannella, editors, *ICALP*, volume 3142 of *LNCS*, pages 408–419. Springer, 2004.

[4] Robert Dąbrowski and Wojciech Plandowski. On word equations in one variable. *Algorithmica*, 60(4):819–828, 2011.

[5] Artur Jeż. Compressed membership for NFA (DFA) with compressed labels is in NP (P). In Christoph Dürr and Thomas Wilke, editors, *STACS*, volume 14 of *LIPIcs*, pages 136–147. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2012.

[6] Artur Jeż. Faster fully compressed pattern matching by recompression. In Artur Czumaj, Kurt Mehlhorn, Andrew Pitts, and Roger Wattenhofer, editors, *ICALP*, volume 7391 of *LNCS*, pages 533–544. Springer, 2012.

[7] Artur Jeż. Recompression: a simple and powerful technique for word equations. *CoRR*, 1203.3705, 2012. accepted for STACS 2013.

[8] Artur Jeż. Approximation of grammar-based compression via recompression. *CoRR*, 1301.5842, 2013. submitted.

[9] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[10] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir and Gad M. Landau, editors, *CPM*, volume 2089 of *Lecture Notes in Computer Science*, pages 181–192. Springer, 2001.

[11] Markku Laine and Wojciech Plandowski. Word equations with one unknown. *Int. J. Found. Comput. Sci.*, 22(2):345–375, 2011.

[12] Markus Lohrey and Christian Mathissen. Compressed membership in automata with compressed labels. In Alexander S. Kulikov and Nikolay K. Vereshchagin, editors, *CSR*, volume 6651 of *LNCS*, pages 275–288. Springer, 2011.

[13] G. S. Makanin. The problem of solvability of equations in a free semigroup. *Matematicheskii Sbornik*, 2(103):147–236, 1977. (in Russian).

[14] Kurt Mehlhorn, R. Sundar, and Christian Uhrig. Maintaining dynamic sequences under equality tests in polylogarithmic time. *Algorithmica*, 17(2):183–198, 1997.

[15] S. Eyono Obono, Pavel Goralcik, and M. N. Maksimenko. Efficient solving of the word equations in one variable. In Igor Prívara, Branislav Rovan, and Peter Ruzicka, editors, *MFCS*, volume 841 of *LNCS*, pages 336–341. Springer, 1994.

[16] Wojciech Plandowski. Satisfiability of word equations with constants is in NEXPTIME. In *STOC*, pages 721–725, 1999.

[17] Wojciech Plandowski. Satisfiability of word equations with constants is in PSPACE. *J. ACM*, 51(3):483–496, 2004.

[18] Wojciech Plandowski. An efficient algorithm for solving word equations. In Jon M. Kleinberg, editor, *STOC*, pages 467–476. ACM, 2006.

[19] Wojciech Plandowski and Wojciech Rytter. Application of Lempel-Ziv encodings to the solution of words equations. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *LNCS*, pages 731–742. Springer, 1998.

[20] Hiroshi Sakamoto. A fully linear-time approximation algorithm for grammar-based compression. *J. Discrete Algorithms*, 3(2-4):416–430, 2005.

Max Planck Institute für informatik, Saarbrücken, Germany, on leave from: Institute of Computer Science, University of Wrocław, Wrocław, Poland